Intel Kernel Virtual Interface Provider Library (KVIPL) Addendum

DRAFT
March 25, 1999

DISCLAIMERS

Table of Contents

1. Introduction

1.1. Overview

The Intel Kernel VI Provider Library (KVIPL) Addendum describes the kernel interface to the VI Architecture. This document should be used in conjunction with the Version 1.0 Intel VI Architecture Developer's Guide and the 1.0 VI Architecture Specification. KVIPL is based on the VIPL defined in the Version 1.0 Intel VI Architecture Developer's Guide, but is not identical to VIPL. Kernel-based applications have different requirements that resulted in changes and extensions to the interface.

1.2. Terminology

1.2.1. Acronyms and Abbreviations

API Application Programming Interface. A collection of function calls exported by libraries and/or services.

IHV Independent Hardware Vendor. Any vendor providing hardware. Used synonymously at times with VI Hardware Vendor.

MTU Maximum Transfer Unit. The largest frame length that may be sent on a physical medium.

NIC Network Interface Controller. A NIC provides an electro-mechanical attachment of a computer to a network. Under program control, a NIC copies data from memory to the network medium, transmission, and from the medium to memory, reception, and implements a unique destination for messages traversing the network.

OSV Operating System Vendor. The software manufacturer of the operating system that is running on the node under discussion.

QOS Quality of Service. Metrics that predict the behavior, speed and latency of a given network connection.

SAN System Area Network. A high-bandwidth, low-latency network interconnecting nodes within a distributed computer system.

VM Virtual Memory. The address space available to a process running in a system with a memory management unit (MMU). The virtual address space is usually divided into pages, each consisting of 2^N bytes. The bottom N address bits (the offset within a page) are left unchanged, indicating the offset within a page, and the upper bits give a (virtual) page number that is mapped by the MMU to a physical page address. This is recombined with the offset to give the address of a location in physical memory.

1.2.2. Industry Terms

Data Payload The amount of data, not including any control or header information, that can be carried in one packet.

Frame One unit of data encapsulated by a physical network protocol header and/or trailer. The header generally provides control and routing information for directing the frame through the network fabric. The trailer generally contains control and CRC data for ensuring packets are not delivered with corrupted contents.

Link A full duplex channel between any two network fabric elements, such as nodes, routers or switches.

Network Fabric
The collection of routers, switches, connectors, and cables that connects a set of nodes.

Message An application-defined unit of data interchange. A primitive unit of communication between cooperating sequential processes.

Message Latency
The elapsed time from the initiation of a message send operation until the receiver is notified that the entire message is present in its memory.

Message Overhead
The sum of the times required to initiate transmission of a message, notify the receiver that the message is available, and the non-bandwidth dependent latencies (e.g. time for a NIC to process data) incurred in moving a message from the source to the destination.

Node A computer attached by a NIC to one or more links of a network, and forming the origin and/or destination of messages within the network.

Packet A primitive unit of data interchange between nodes, comprised of a set of data segments transmitted in an ordered stream. A packet may be sent as a single frame, or may be fragmented into smaller units (cells) such that cells for various packets may be interleaved in the fabric but the transmission order of cells for a packet is preserved and manifest as a contiguous unit at a receiving node.

Server The class of computers that emphasize I/O connectivity and centralized data storage capacity to support the needs of other, typically remote, client computers.

Workstation, or Client
The class of computers that emphasize numerical and/or graphic performance and provide an interface to a human being.

1.2.3. VI Architecture Terms

The following terms were introduced in the VI Architecture Specification.

Address Segment
The second of the three segments that comprise a remote-DMA operation Descriptor, specifying the memory region to access on the target.

Communication Memory
Any region of a process' memory that is registered with the VI Provider to serve for storage of Descriptors and/or as communication buffers; i.e., any region of a process' memory that will be accessed by the VI NIC.

Connection An association between a pair of VIs such that messages sent using either VI arrives at the other VI. A VI is either unconnected, or connected to one and only one other VI.

Control Segment
The first component of a Descriptor containing information regarding the type of VI NIC data movement operation to be performed, the status of a completed VI NIC data movement operation, and the location of the next Descriptor on a Work Queue.

Completion Queue
A queue containing information about completed Descriptors. Used to create a single point of completion notification for multiple queues.

Completion Queue Entry
A single data structure on a Completion Queue that describes a completed Descriptor. This entity contains sufficient information to determine the queue that holds the completed Descriptor.

Data Segment A component of a Descriptor specifying one memory region for the VI NIC to use as a communication buffer.

Descriptor A data structure recognized by the VI NIC that describes a data movement request. A Descriptor is organized as a list of segments. A Descriptor is comprised of a control segment followed by an optional address segment and an arbitrary number of data segments. The data segments describe a communication buffer gather or scatter list for a VI NIC data movement operation.

Doorbell A mechanism for a process to notify the VI NIC that work has been placed on a Work Queue. The Doorbell mechanism must be protected by the operating system—i.e., for address protection, only the operating system should be able to establish a Doorbell—and the VI NIC must be able to identify the owner of a VI by the use of its Doorbell.

Immediate Data
Data contained in a Descriptor that is sent along with the data to the remote node and placed in the remote node's pre-posted Receive Queue Descriptor.

Kernel Agent A component of the operating system required by the VI Architecture that subsumes the role of the device driver for the VI NIC and includes the kernel software needed to register communication memory and manage VIs.

KVIPL An implementation of the VI User Agent for kernel applications. KVIPL is an acronym for Kernel Virtual Interface Provider Library.

Memory Handle
A programmatic construct that represents a process's authorization to specify a memory region to the VI NIC. A memory handle is created by the VI Kernel Agent when a process registers communication memory. A process must supply a corresponding Memory Handle with any virtual address to qualify it to the VI NIC. The VI NIC will not perform an access to a virtual address if the supplied memory handle does not agree with the memory region containing the virtual address or if the memory region is registered to a process other than the process that owns a Virtual Interface (VI).

Memory Protection Attributes
The access rights for RDMA granted to VIs and to Memory Regions.

Memory Protection Tag
A unique identifier generated by the VI Provider for use by the VI Consumer. Memory Protection Tags are associated with VIs and Memory Regions to define the access permission the VI has to a memory region.

Memory Region
An arbitrary sized region of a process's virtual address space registered as communication memory such that it can be directly accessed by the VI NIC.

Memory Registration
The act of creating a memory region. The memory registration operation returns a Memory Handle that the process is required to provide with any virtual address within the memory region.

VI NIC Address
: The logical network address of the VI NIC. This address is assigned to a VI NIC by the operating system and allows processes within a network to identify a remote node with respect to a VI NIC attachment of the remote node to the network.

NIC Handle
: A programmatic construct representing a process's authorization to perform communication operations using a local VI NIC.

Peer
: A generic term for the process at the other end of a connection.

Post
: To place a Descriptor on a VI Work Queue.

RDMA
: Remote Direct Memory Access. A Descriptor operation whereby data in a local gather or scatter list is moved directly to or from a memory region on a remote node. A process authorizes remote access to its memory by creating a VI with remote-DMA operations enabled, connecting it to a remote VI, and making the memory handle for the memory region to be shared available to the peer that will perform the remote-DMA operation. There are two remote-DMA operations: write and read.

Receive Queue
: One of the two queues associated with a VI. This queue contains Descriptors that describe where to place incoming data.

Reliable Delivery
: The second of three communication reliability levels. Guarantees that all data submitted for transfer will arive at its destination exactly once, intact and in the order submitted, in the absence of errors. The VI Provider must deliver an error to the VI Consumer if a transfer is lost, corrupted or delivered out of order.

Reliable Reception
: The highest communication reliability level. A Descriptor is completed with a successful status only when the data has been delivered into the target memory location. If an error occurs that prevents a successful (in-order, intact and exactly once) delivery of the data into the target memory, the error is reported through the Descriptor status. Otherwise, a Reliable Reception VI behaves like a Reliable Delivery VI.

Send Queue
: One of the two queues associated with a VI. This queue contains Descriptors that describe the data to be transmitted.

Unreliable Delivery
: The lowest communication reliability level. This level guarantees that a Send or RDMA Write is delivered at most once to the receiving VI and corrupted transfers are detected on the receiving side. Sends and RDMA Writes may be lost on an Unreliable Delivery VI. In addition, requests are not guaranteed to be delivered to the receiver in the order submitted by the sender. However, the order must adhere to the Descriptor processing ordering rules.

User Agent
: A software component that enables an Operating System communication facility to utilize a particular VI Provider. The VI User Agent abstracts the details of the underlying VI NIC hardware in accordance with an interface defined by the Operating System communication facility.

VI
: Virtual Interface. An interface between a VI NIC and a process allowing a VI NIC direct access to the process' memory. A VI consists of a pair of Work Queues—one for send operations and one for receive operations. The queues store a Descriptor between the time it is posted and the time it is Done. A pair of VIs are associated using the connect operation to allow packets sent at one VI to be received at the other.

VI Address The logical name for a VI. The VI address identifies a remote end-point to be associated with a local end-point using the connect-VI operation.

VI Application An application that uses the primitives provided by the VI User Agent.

VI Consumer A software process that communicates using a Virtual Interface. The VI Consumer typically consists of an application program, an Operating System communications facility, and a VI User Agent.

VI Handle A programmatic construct that represents a processes authorization to perform operations on a specific VI. A VI handle is returned by the operation that creates the VI and is supplied as an identifier parameter to the other VI operations.

VI Hardware Vendor Anyone who produces a VI Architecture enabled NIC implementation. The vendor is responsible for providing the VI NIC, VI Kernel Agent and the VI User Agent.

VI NIC A Network Interface Controller that complies with the VI Architecture Specification.

VIPL An implementation of the VI User Agent. VIPL is an acronym for Virtual Interface Provider Library.

VI Provider The combination of a VI NIC and a VI Kernel Agent. Together, these two components instantiate a Virtual Interface.

Work Queue A posted list of Descriptors being processed by a VI NIC. Every VI has two Work Queues: a send queue and a receive queue. The combination of the Work Queue selected by a post operation and the operation type indicated by the Descriptor determine the exact type of data movement that the VI NIC will perform.

2. Kernel Virtual Interface Provider Library (KVIPL)

2.1. Overview

This section describes a reference interface to the VI Architecture for kernel applications, referred to as KVIPL. The material is presented in the form of groups of related routines, followed by definitions of data structures, constants and error codes. Semantic clarifications on specific routines are provided at the end of respective sections, whenever deemed necessary.

Note: it is the responsibility of the client to ensure handles and pointers passed to KVIPL are valid. KVIPL does not check the validity of these parameters.

3. KVIPL Calls

3.1. Hardware Connection

3.1.1. KvipOpenNic

Synopsis

```
KVIP_RETURN
        KvipOpenNic(
                IN      const KVIP_WCHAR    *DeviceName,
                OUT     KVIP_NIC_HANDLE     NicHandle,
                IN      KVIP_WAIT_OBJECT    *ErrorNotificationWaitObject
        )
```

Parameters

DeviceName: Symbolic name of the device (VI Provider instance) associated with the NIC.

NicHandle: Handle returned. The handle is used with the other functions to specify a particular instance of a VI NIC.

ErrorNotificationWaitObject: The KVIP_WAIT_OBJECT that is signaled by KVIPL when an asynchronous error occurs.

Description

KvipOpenNic associates a process with a VI NIC, and provides a NIC handle to the VI Consumer. The NIC handle is used in subsequent functions in order to specify a particular NIC. A process is allowed to open the same VI NIC multiple times. Each time a process calls *KvipOpenNic* with the same device name, a different NIC handle is returned that references the same NIC.

The ErrorNotificationWaitObject parameter provides a means for KVIPL client applications to be notified of asynchronous errors. Asynchronous errors cannot be reported back directly in the Descriptor. The application has the option to wait, poll or use polling and waiting in combination for error Descriptors to complete. The application can wait for the wait object to be signaled using standard kernel synchronization routines. The application polls to check for errors by calling *KvipErrorNotificationDone*. It is the responsibility of the calling routine to manage the memory for the ErrorNotificationWaitObject. That is, the calling routine must allocate the memory for the object and free the memory when the NIC is closed.

The *KvipRearmErrorNotification* function resets the ErrorNotificationWaitObject to allow the application to wait on the object again. A recommended programming model is to call *KvipRearmErrorNotification* to reset the wait object after processing all errors from the associated NIC.

If ErrorNotificationWaitObject is NULL, a KVIP_INVALID_PARAMETER error will be returned.

Asynchronous errors are those errors that cannot be reported back directly in the Descriptor. The following is a list of possible asynchronous errors:

- Post Descriptor error - This error occurs under the following conditions:
  - The virtual address and memory handle of the Descriptor was not valid when the Descriptor was posted.
  - The Next Address and/or Next Handle field was inadvertently modified after the Descriptor was posted.
  - The Descriptor address was not aligned on a 64-byte boundary.

- Connection Lost – The connection on a VI was lost and the associated VI is in the Error state.
- Receive Queue Empty – An incoming packet was dropped because the receive queue was empty.
- VI Overrun – The VI Consumer attempted to post too many Descriptors to a Work Queue of a VI.
- RDMA Write Protection Error – A protection error was detected on an incoming RDMA Write operation. If the RDMA write operation contained immediate data, this status would be reported in the associated Descriptor, and not via the asynchronous error mechanism.
- RDMA Write Data Error – A data corruption error was detected on an incoming RDMA Write operation. If the RDMA write operation contained immediate data, this status would be reported in the associated Descriptor, and not via the asynchronous error mechanism.
- RDMA Write Packet Abort – Indicates a partial packet was detected on an incoming RDMA Write operation. If the RDMA write operation contained immediate data, this status would be reported in the associated Descriptor, and not via the asynchronous error mechanism.
- RDMA Transport Error - A transport error was detected on an incoming RDMA operation that does not consume a Descriptor.
- RDMA Read Protection Error – A protection error was detected on an incoming RDMA Read operation.
- Completion Protection Error - When reporting completion, this could result from a user de-registering a memory region containing a Descriptor after the Descriptor was read by the hardware but before completion status was written.

  This error can also result if a Completion Queue becomes inaccessible to the hardware. In this case, an error will be generated for each VI that was associated with the Completion Queue. Note that the status (Done or Done with Errors) of the Descriptors on work queues associated with the Completion Queue may have already been written.
- Catastrophic Error - The hardware has failed or has detected a fatal configuration problem.

  Post Descriptor Error, VI Overrun, Completion Protection Error and Catastrophic Error are catastrophic hardware errors.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_ERROR_RESOURCE – An error was detected due to insufficient resources.

KVIP_INVALID_PARAMETER – Device name does not exist.

3.1.2. KvipCloseNic

Synopsis

```
KVIP_RETURN
        KvipCloseNic(
                IN      KVIP_NIC_HANDLE     NicHandle
        )
```

Parameters

NicHandle: The NIC handle.

Description

KvipCloseNic removes the association between the calling process and the VI NIC that was established via the corresponding *KvipOpenNic* function.

When a VI NIC is closed, it is the responsibility of the VI Provider to clean up all resources associated with that NIC instance. This includes any resources allocated by the library and threads started on behalf of the NIC.

Returns

KVIP_SUCCESS – Operation completed successfully.

3.2. Endpoint Creation and Destruction

3.2.1. KvipCreateVi

Synopsis

```
KVIP_RETURN
        KvipCreateVi(
                IN      KVIP_NIC_HANDLE     NicHandle,
                IN      KVIP_VI_ATTRIBUTES *ViAttribs,
                IN      KVIP_CQ_HANDLE      SendCQHandle,
                IN      KVIP_CQ_HANDLE      RecvCQHandle,
                IN      KVIP_WAIT_OBJECT    *SendWaitObject,
                IN      KVIP_WAIT_OBJECT    *RecvWaitObject,
                OUT     KVIP_VI_HANDLE      *ViHandle
        )
```

Parameters

NicHandle: Handle of the associated VI NIC.

ViAttribs: The initial attributes to set for the new VI.

SendCQHandle: The handle of a Completion Queue. If a valid handle, the send Work Queue of this VI will be associated with the Completion Queue. If NULL, the send queue is not associated with any Completion Queue.

RecvCQHandle: The handle of a Completion Queue. If valid, the receive Work Queue of this VI will be associated with the Completion Queue. If NULL, the receive queue is not associated with any Completion Queue.

SendWaitObject: A synchronization object that is signaled by KVIPL when a completed descriptor is present on the send Work Queue of this VI. SendWaitObject can only be used if SendCQHandle for this VI is NULL. That is, the send Work Queue of this VI is not associated with any completion queue.

RecvWaitObject: A synchronization object that is signaled by KVIPL when a completed descriptor is present on the receive Work Queue of this VI. RecvWaitObject can only be used if RecvCQHandle for this VI is NULL. That is, the receive Work Queue of this VI is not associated with any completion queue.

ViHandle: The handle for the newly created VI instance.

Description

KvipCreateVi creates an instance of a Virtual Interface to the specified NIC.

The ViAttribs input parameter specifies the initial attributes for this VI instance.

The SendCQHandle and RecvCQHandle parameters allow the caller to associate the Work Queues of this VI with a Completion Queue. If a Work Queue is associated with a Completion Queue, the calling process cannot wait on that queue via *KvipSendWait* or *KvipRecvWait*.

The SendWaitObject and RecvWaitObject parameters provide a means for KVIPL client applications to be notified when completed descriptors are available on the VI's send or receive Work Queues respectively. If the VI Work Queue is associated with a completion queue, then CQWaitObject must be used instead of SendWaitObject or RecvWaitObject. The application has the option to wait, poll or use polling and waiting in combination.. The application can wait for the wait objects to be signaled using standard kernel synchronization routines. The application polls by use checking the completion status of the appropriate Work Queue using the *KvipSendDone* or the *KvipRecvDone* function. If the application only polls for completions, SendWaitObject and RecvWaitObject can be set to NULL

The *KvipRearmSend* and *KvipRearmRecv* functions reset SendWaitObject and RecvWaitObject respectively to allow the application to wait on the object again after a signal. A recommended programming model is to call *KvipRearmSend* or *KvipRearmRecv* to reset the appropriate wait object after processing all of the completed descriptors from the associated Work Queue.

When a new instance of a VI is created, it begins in the Idle state.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_ERROR_RESOURCE – Insufficient resources.

KVIP_INVALID_RELIABILITY_LEVEL – The requested reliability level attribute was invalid or not supported.

KVIP_INVALID_MTU – The maximum transfer size attribute was invalid or not supported.

KVIP_INVALID_QOS – The quality of service attribute was invalid or not supported.

KVIP_INVALID_PTAG – The protection tag attribute was invalid.

KVIP_INVALID_RDMAREAD – The attributes requested support for RDMA Read, but the VI Provider does not support it.

3.2.2. KvipDestroyVi

Synopsis

```
KVIP_RETURN
    KvipDestroyVi(
        IN      KVIP_VI_HANDLE      ViHandle
    )
```

Parameters

ViHandle: The handle of the VI instance to be destroyed.

Description

KvipDestroyVi tears down a Virtual Interface. A VI instance may only be destroyed if the VI is in the Idle state and all Descriptors on its work queues have been de-queued, otherwise an error is returned to the caller. Use of the destroyed handle in any subsequent operation will fail.

Returns

KVIP_SUCCESS – Operation completed successfully

KVIP_INVALID_STATE – The VI is not in the Idle state or there are still Descriptors posted on the work queues.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.3. Connection Management

3.3.1. KvipConnectServerRequest

Synopsis

```
KVIP_RETURN
    KvipConnectServerRequest(
        IN    KVIP_NIC_HANDLE     NicHandle,
        IN    KVIP_NET_ADDRESS    *LocalAddr,
        IN    KVIP_ULONG          Timeout,
        IN    KVIP_WAIT_OBJECT    *WaitObject,
        OUT   KVIP_CONN_HANDLE    *ConnHandle
    )
```

Parameters

NicHandle: Handle for an instance of a VI NIC.

LocalAddr: Local network address. Only the discriminator portion of the net address is used to determine if the request matches. The host address portion must match the local NIC address.

Timeout: The count, in milliseconds, that the server will look for valid incoming VI connection requests from a client. KVIP_INFINITE if no time-out is desired. A timeout of zero is invalid.

WaitObject: A synchronization object that a server thread waits on for completion of the connection request. A NULL WaitObject indicates the server application will use a polling model to check for completion of the connection request.

ConnHandle: A handle to an opaque connection object subsequently used in calls to *KvipConnectServerDone*.

Description

KvipConnectServerRequest is called by the server to look for incoming VI connection requests from a client.

If WaitObject is NULL, the server application must call *KvipConnectServerDone* to poll for the completion status and get the attributes of the client connection. If WaitObject is non-NULL, the server can either wait on the WaitObject itself or have another thread wait for it. The server application can periodically poll to check the completion status of this connect request using *KvipConnectServerDone*. For any WaitObject setting, *KvipConnectServerDone* must be called to complete the connection operation and get the attributes of the client connection.

If the timeout value specified in the Timeout parameter expires before a valid connection request from a client arrives, a subsequent call to *KvipConnectServerDone* fails and returns a KVIP_TIMEOUT error. If a timeout value of zero is specified, KVIP_INVALID_PARAMETER is returned.

Returns

KVIP_SUCCESS - The connection request was successfully posted by the server.

KVIP_INVALID_PARAMETER - One of the parameters was invalid.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.3.2. KvipConnectServerDone

Synopsis

```
KVIP_RETURN
    KvipConnectServerDone(
        IN    KVIP_CONN_HANDLE  ConnHandle,
        OUT   KVIP_NET_ADDRESS  *RemoteAddr,
        OUT   KVIP_VI_ATTRIBUTES *RemoteViAttribs
    )
```

Parameters

ConnHandle: A handle to an opaque connection object created by *KvipConnectServerRequest*.

RemoteAddr: The remote network address (host address and discriminator) that is requesting a connection. The value of the network address returned is the same as the LocalAddr parameter supplied to the matching *KvipConnectClientRequest*.

RemoteViAttribs: The attributes of the remote VI endpoint that is requesting the connection.

Description

KvipConnectServerDone is called by the server to check the status of a previous *KvipConnectServerRequest*. The call returns a KVIP_SUCCESS status when a connection has been established with a client.

Returns

KVIP_SUCCESS - The connection establishment was successful.

KVIP_TIMEOUT - Returned when the Timeout specified in *KvipConnectServerRequest* expired before a successful connection could be established.

KVIP_NOT_DONE - The connection has not been established and the Timeout has not expired.

3.3.3. KvipConnectClientRequest

Synopsis

```
KVIP_RETURN
    KvipConnectClientRequest(
        IN    KVIP_VI_HANDLE      ViHandle,
        IN    KVIP_NET_ADDRESS    *LocalAddr,
        IN    KVIP_NET_ADDRESS    *RemoteAddr,
        IN    KVIP_ULONG          Timeout,
        IN    KVIP_WAIT_OBJECT    *WaitObject
    )
```

Parameters

ViHandle: Handle for the local VI endpoint.

LocalAddr: Local network address. The local address is used solely for naming purposes and is not used for matching by *KvipConnectClientDone*. The host address portion must match the local NIC address. The discriminator portion is passed unchanged to the RemoteAddr structure returned from the matching *KvipConnectClientDone.*

RemoteAddr: The remote network address. The remote network address must contain both the host address and the discriminator.

Timeout: The count, in milliseconds, that the client request remains active. KVIP_INFINITE if no time-out is desired. A timeout of zero is invalid.

RemoteAddr: The remote network address (host address and discriminator) that is requesting a connection. The value of the network address returned is the same as the LocalAddr parameter supplied to the matching *KvipConnectServerRequest*.

WaitObject: An object that a client thread waits on for completion of the connection request. A NULL WaitObject indicates the client will use a polling model to check for completion of the connection request.

Description

KvipConnectClientRequest is called by the client to post a request to establish a new connection with a remote server.

If WaitObject is NULL, the client application must call *KvipConnectClientDone* to poll for the completion status and to get the attributes of the connection. If WaitObject is non-NULL, the client can either wait on the WaitObject itself or have another thread wait for it. The client application can periodically poll to check the completion status of this connect request using *KvipConnectClientDone*. For any WaitObject setting, *KvipConnectClientDone* must be called to get the attributes of the remote VI endpoint.

If the timeout value specified in the Timeout parameter expires before a valid connection is established, a subsequent call to *KvipConnectClientDone* fails and returns a KVIP_TIMEOUT error. A timeout value of zero is invalid and results in a KVIP_INVALID_PARAMETER error.

Returns

KVIP_SUCCESS - The connection request was successfully posted by the client.

KVIP_INVALID_PARAMETER - One of the parameters is invalid.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.3.4. KvipConnectClientDone

Synopsis

```
KVIP_RETURN
        KvipConnectClientDone(
                IN      KVIP_VI_HANDLE        ViHandle,
                OUT     KVIP_VI_ATTRIBUTES *RemoteViAttribs
        )
```

Parameters

ViHandle: Handle for the local VI endpoint.

RemoteViAttribs: The attributes of the remote VI endpoint that is requesting the connection.

Description

KvipConnectClientDone is called by the client to check the status of a previous *KvipConnectClientRequest*. The call returns a KVIP_SUCCESS status when a connection has been established with the server.

Returns

KVIP_SUCCESS - The connection establishment was successful.

KVIP_REJECT - The connection was rejected.

KVIP_TIMEOUT - Returned when the Timeout specified in *KvipConnectClientRequest* expired before a successful connection was established.

KVIP_INVALID_PARAMETER - A timeout of zero was specified.

KVIP_NOT_DONE - The connection has not been established and the Timeout has not expired.

KVIP_NO_MATCH - The server is up and is not waiting for a connection request with the specified discriminator.

KVIP_NOT_REACHABLE - A network partition was detected.

3.3.5. KvipConnectAccept

Synopsis

```
KVIP_RETURN
    KvipConnectAccept(
        IN    KVIP_CONN_HANDLE  ConnHandle,
        IN    KVIP_VI_HANDLE    ViHandle
    )
```

Parameters

ConnHandle: A handle to an opaque connection object created by *KvipConnectServerRequest*.

ViHandle: Instance of a local VI endpoint.

Description

KvipConnectAccept is used to accept a connection request and associate the connection with a local VI endpoint. This function is called on the server side of the client/server connection model.

The caller passes in the handle of an Idle VI endpoint to associate with the connection request. If the attributes of the local VI endpoint conflict with those of the remote endpoint, *KvipConnectAccept* will fail. It is the function of the VI Provider to determine if the connection should succeed based on the attributes of the two endpoints. Note: The VI attributes that must match when establishing a connection are ReliabilityLevel, MaxTransferSize and QoS.

If *KvipConnectAccept* fails, no explicit notification is sent to the remote end. The caller may choose to modify the attributes of the local VI endpoint, and try again. In order to reject a connection request, the VI Consumer must explicitly reject the connection request via the *KvipConnectReject* function.

Returns

KVIP_SUCCESS – The connection was successfully established.

KVIP_INVALID_ RELIABILITY_LEVEL – The reliability level attribute of the local endpoint conflicted with the reliability level of the remote VI.

KVIP_INVALID_MTU – The maximum transfer size attribute of the local endpoint conflicted with the maximum transfer size of the remote endpoint.

KVIP_INVALID_QOS – The quality of service attribute was invalid, or conflicted with the remote endpoint.

KVIP_TIMEOUT - The connection could not be completed, possibly due to a timeout failure on the matching *KvipConnectClientRequest*.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

KVIP_INVALID_STATE - The specified VI endpoint is not in the Idle state.

KVIP_NOT_REACHABLE - A network partition was detected.

3.3.6. KvipConnectReject

Synopsis

```
KVIP_RETURN
    KvipConnectReject(
        IN      KVIP_CONN_HANDLE  ConnHandle
    )
```

Parameters

ConnHandle: A handle to an opaque connection object created by *KvipConnectServerRequest.*

Description

KvipConnectReject is used to reject a connection request. This function is called on the server side of the client/server connection model. Notification is sent to the remote end that the associated connection request was rejected.

Returns

KVIP_SUCCESS – The operation completed successfully.

KVIP_ERROR_RESOURCE - The operation failed due insufficient resources.

KVIP_NOT_REACHABLE - A network partition was detected.

3.3.7. KvipDisconnect

Synopsis

```
KVIP_RETURN
    KvipDisconnect(
        IN      KVIP_VI_HANDLE      ViHandle
    )
```

Parameters

ViHandle: Instance of a connected Virtual Interface endpoint.

Description

KvipDisconnect is used to terminate a connection. When the local endpoint is disconnected, it stops processing of all posted Descriptors, all pending (not completed) Descriptors are marked completed because of disconnection with a Descriptor Flushed error, and the local endpoint transitions to the Idle state. When the remote endpoint causes a connection to terminate by closing the endpoint or by calling *KvipDisconnect*, an asynchronous error callback is generated

indicating the disconnected connection. The local client should call *KvipDisconnect* to reset the disconnected connection from the Error state to the Idle state.

KvipDisconnect can be called in any VI state to cause pending Descriptors on the VI to be completed with the Descriptor Flushed error bit set and transition the VI to the Idle state. Specifically, *KvipDisconnect* can be called in the Idle state to force pending receive Descriptors to be completed in error so they can be de-queued prior to calling *KvipDestroyVi.*

Returns

KVIP_SUCCESS – The disconnect operation was successful.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.3.8. KvipConnectPeerRequest

Synopsis

```
KVIP_RETURN
        KvipConnectPeerRequest(
                IN      KVIP_VI_HANDLE      ViHandle,
                IN      KVIP_NET_ADDRESS    *LocalAddr,
                IN      KVIP_NET_ADDRESS    *RemoteAddr,
                IN      KVIP_ULONG          Timeout,
                IN      KVIP_WAIT_OBJECT    *WaitObject
        )
```

Parameters

ViHandle:	Handle for the local VI endpoint.
Local Addr:	Local network address. The local address is used solely for administrative purposes and is not used for matching connection requests. The host portion of this network address must match the NIC's address.
RemoteAddr:	The remote network address.
Timeout:	The count, in milliseconds, that specifies the length of time to wait for the connection to complete. KVIP_INFINITE if no time-out is desired. A timeout of zero is invalid.
WaitObject:	An object that the application thread waits on for completion of the connection request. A NULL WaitObject indicates the application will use a polling model to check for completion of the connection request.

Description

KvipConnectPeerRequest posts a request that a connection be established between the local VI endpoint and a remote VI endpoint. The user specifies local and remote network addresses as part of the connection request. This call returns as soon as the connection request is initiated. The caller of *KvipConnectPeerRequest* can check the status of the connection request by calling *KvipConnectPeerDone*.

If WaitObject is NULL, the peer application must call *KvipConnectPeerDone* to poll for the completion status and to get the attributes of the connection. If WaitObject is non-NULL, the application can either wait on the WaitObject itself or have another thread wait for it. The application can periodically poll to check the completion status of this connect request using *KvipConnectPeerDone*. For any WaitObject setting, *KvipConnectPeertDone* must be called to complete the connection operation and get the attributes of the remote VI endpoint.

If a connection is successfully established, the local address is bound to the local VI endpoint and the attributes of the remote VI endpoint are returned to the caller when the connect completion status is delivered. If a connection cannot be established before the specified Timeout period, the WaitObject is signaled (when used) and a KVIP_TIMEOUT error is returned in the return status of *KvipConnectPeerDone*. A timeout value of zero is invalid and results in a KVIP_INVALID_PARAMETER error.

If the host portion of the LocalAddr parameter does not match the local NIC address (specified in the LocalNicAddress field of the NicAttributes structure), then a KVIP_INVALID_PARAMETER error is returned.

Returns

KVIP_SUCCESS - The connection request was queued.

KVIP_ERROR_RESOURCE - The request failed due to insufficient resources.

KVIP_INVALID_STATE - The specified VI endpoint is not in the Idle state.

KVIP_INVALID_PARAMETER - One of the parameters was invalid.

3.3.9. KvipConnectPeerDone

Synopsis

```
KVIP_RETURN
    KvipConnectPeerDone(
        IN   KVIP_VI_HANDLE          ViHandle,
        OUT  KVIP_VI_ATTRIBUTES      *RemoteViAttribs
    )
```

Parameters

ViHandle:	Handle for the local VI endpoint.
RemoteVIAttribs:	The attributes of the remote VI endpoint if the connection was successfully completed.

Description

KvipConnectPeerDone is called to determine the results of a previously posted *KvipConnectPeerRequest* call on the specified VI handle, without blocking the calling thread.

If a connection is successfully established, the attributes of the remote endpoint are returned to the caller. The attributes of the remote endpoint allow the caller to determine whether/which RDMA operations can be executed on the resulting connection. Note: The VI attributes that must match are ReliabilityLevel, MaxTransferSize and QoS.

If the connection was not successfully established within the Timeout period specified in *KvipConnectPeerRequest*, a KVIP_TIMEOUT error is returned. *KvipConnectPeerDone* returns KVIP_NOT_DONE if the connection operation is still in progress.

Returns

KVIP_SUCCESS – The connection completed successfully. Attributes of the remote endpoint are returned through the second function parameter.

KVIP_INVALID_ RELIABILITY_LEVEL – The reliability level attribute of the local endpoint conflicted with the reliability level of the remote VI.

KVIP_INVALID_MTU – The maximum transfer size attribute of the local endpoint conflicted with the maximum transfer size of the remote endpoint.

KVIP_INVALID_QOS – The quality of service attribute was invalid, or conflicted with the remote endpoint.

KVIP_TIMEOUT – The connection request timeout expired before a successful connection completion

KVIP_NOT_DONE - The connection operation is still in progress.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

KVIP_INVALID_STATE - The specified VI endpoint is not in the Pending Connect state.

KVIP_NOT_REACHABLE - A network partition was detected.

3.4. Memory protection and registration

3.4.1. KvipCreatePtag

Synopsis

```
KVIP_RETURN
        KvipCreatePtag(
                IN      KVIP_NIC_HANDLE             NicHandle,
                OUT     KVIP_PROTECTION_HANDLE      *Ptag
        )
```

Parameters

NicHandle: The NIC handle associated with the protection tag.

Ptag: The new protection tag.

Description

The *KvipCreatePtag* function creates a new protection tag for the calling process. The protection tag is subsequently associated with VI endpoints via the *KvipCreateVi* function, as well as memory regions via the *KvipRegisterMem* function. A process may request multiple protection tags.

For all memory references by the VI Provider, including Descriptors and message buffers, the protection tag of the VI instance, and the memory region, must match in order to pass the memory protection check.

The Protection Tag is an element in the VI attributes data structure and the Memory Region Attributes data structure. The protection tag of a memory region and/or a VI can be changed by changing their attributes.

Returns

KVIP_SUCCESS – The memory protection tag was successfully created.

KVIP_ERROR_RESOURCE – The operation failed due to insufficient resources.

3.4.2. KvipDestroyPtag

Synopsis

```
KVIP_RETURN
        KvipDestroyPtag(
                IN      KVIP_NIC_HANDLE             NicHandle,
                IN      KVIP_PROTECTION_HANDLE      Ptag
        )
```

Parameters

NicHandle: The NIC handle associated with the protection tag.

Ptag: The protection tag.

Description

The *KvipDestroyPtag* function destroys a protection tag.

If the specified protection tag is associated with either a VI instance or a registered memory region at the time of the call, an error is returned.

Returns

KVIP_SUCCESS – The memory protection tag was successfully destroyed.

KVIP_ERROR_RESOURCE – A VI instance or a registered memory region is still associated with the specified protection tag or the operation failed due to insufficient resources.

3.4.3. KvipRegisterMem

Synopsis

```
KVIP_RETURN
        KvipRegisterMem(
                IN      KVIP_NIC_HANDLE             NicHandle,
                IN      KVIP_PVOID                  VirtualAddress,
                IN      KVIP_ULONG                  Length,
                IN      KVIP_MEM_ATTRIBUTES         *MemAttribs,
                OUT     KVIP_MEM_HANDLE             *MemoryHandle

        )
```

Parameters

NicHandle: Handle for a currently open NIC.

VirtualAddress: Starting address of the memory region to be registered.

Length: The length, in bytes, of the memory region.

MemAttribs: The memory attributes to associate with the memory region.

MemoryHandle: If successful, the new memory handle for the region, otherwise NULL.

Description

KvipRegisterMem allows a process to register a region of memory with a VI NIC. Memory used to hold Descriptors or data buffers must be registered with this function.

The user may specify an arbitrary size region of memory, with arbitrary alignment, but the actual area of memory registered will be registered on page granularity. Registered pages are locked into physical memory.

The memory attributes include the Protection Tag, and the RDMA enable bits that are initially associated with the memory region.

Descriptors and data buffers contained within registered memory can be used by any VI with a matching protection tag that is owned by the process**.** A new memory handle is generated for each region of memory that is registered by a process.

The EnableRdmaWrite memory attribute can be used to ensure that no remote process can modify a region of memory, this could be particularly useful to protect regions of memory that contain Descriptors (control information). The EnableRdmaRead parameter can be used to ensure that no remote process can read a particular region of memory.

Note that the implementation of *KvipRegisterMem* should always check for read-only pages of memory and not allow modification to those pages by the VI Hardware.

A Length parameter of zero will result in a KVIP_INVALID_PARAMETER error.

The contents of the memory region being registered are not altered. The memory region must have been previously allocated by the VI Consumer.

Returns

KVIP_SUCCESS – The memory region was successfully registered.

KVIP_ERROR_RESOURCE – The registration operation failed due to insufficient resources.

KVIP_INVALID_PARAMETER – One of the parameters was invalid.

KVIP_INVALID_PTAG – The protection tag attribute was invalid.

KVIP_INVALID_RDMAREAD – The attributes requested the memory region be enabled for RDMA Read, but the VI Provider does not support it.

3.4.4. KvipDeregisterMem

Synopsis

```
KVIP_RETURN
    KvipDeregisterMem(
        IN    KVIP_NIC_HANDLE    NicHandle,
        IN    KVIP_PVOID         VirtualAddress,
        IN    KVIP_MEM_HANDLE    MemoryHandle
    )
```

Parameters

NicHandle: The handle for the NIC that owns the memory region being de-registered.

VirtualAddress: Address of the region of memory to be de-registered.

MemoryHandle: Memory handle for the region; obtained from a previous call to *KvipRegisterMem*.

Description

KvipDeregisterMem de-registers memory that was previously registered using the *KvipRegisterMem* function and unlocks the associated pages from physical memory. The contents and attributes of the region of virtual memory being de-registered are not altered in any way.

Returns

KVIP_SUCCESS – The memory region was successfully de-registered.

KVIP_INVALID_PARAMETER – One or more of the parameters was invalid.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.4.5. KvipRegisterPhysPages

Synopsis

```
KVIP_RETURN
    KvipRegisterPhysPages(
        IN    KVIP_NIC_HANDLE        NicHandle,
        IN    KVIP_PHYS_ADDRESS      PhysAddress[],
        IN    KVIP_ULONG             NumPhysPages,
        IN    KVIP_MEM_ATTRIBUTES    *MemAttribs,
        OUT   KVIP_MEM_HANDLE        *MemoryHandle
    )
```

Parameters

NicHandle: Handle for a currently open NIC.

PhysAddress: Array of page addresses to register.

NumPhysPages: The number of physical pages to register. This parameter specifies the number of entries in the PhysAddress array.

MemAttribs: The memory attributes to associate with the pages.

MemoryHandle: If successful, the new memory handle for the registered pages, otherwise NULL.

Description

KvipRegisterPhysPages allows a process to register one or more physical memory pages with a VI NIC. Memory used to hold Descriptors or data buffers must be registered with either this function or *KvipRegisterMem*.

The user may specify an arbitrary number of pages to register, specified in the NumPhysPages parameter. The addresses specified in the PhysAddress array must be page aligned or a KVIP_INVALID_PARAMETER error will result. An application referencing a registered physical memory region should use the index into the buffer as the physical address, treating the buffer as if it were virtually contiguous with a starting address of zero. For instance, on a machine with 4K pages, the location 0x300 on the second page maps to the virtual address 0x1300 (4K is 0x1000). This applies to the addresses in descriptors and address passed into other KVIPL calls.

A NumPhysPages parameter of zero will result in a KVIP_INVALID_PARAMETER error.

The contents of the memory pages being registered are not altered. The memory pages must have been previously allocated and pinned by the VI Consumer.

Returns

KVIP_SUCCESS – The memory region was successfully registered.

KVIP_ERROR_RESOURCE – The registration operation failed due to insufficient resources.

KVIP_INVALID_PARAMETER – One of the parameters was invalid.

KVIP_INVALID_PTAG - The protection tag attribute was invalid.

KVIP_INVALID_RDMAREAD - The attributes requested the memory region be enabled for RDMA Read, but the VI Provider does not support it.

3.4.6. KvipDeregisterPhysPages

Synopsis

```
KVIP_RETURN
        KvipDeregisterMem(
                IN      KVIP_NIC_HANDLE     NicHandle,
                IN      KVIP_MEM_HANDLE     MemoryHandle
        )
```

Parameters

NicHandle: The handle for the NIC that owns the memory region being de-registered.

MemoryHandle: Memory handle for the region; obtained from a previous call to *KvipRegisterPhysPages.*

Description

KvipDeregisterPhysPages de-registers memory that was previously registered using the *KvipRegisterPhysPages* function. The contents and attributes of the region of physical memory being de-registered are not altered in any way.

Returns

KVIP_SUCCESS – The memory region was successfully de-registered.

KVIP_INVALID_PARAMETER – One or more of the parameters was invalid.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.5. Data transfer and completion operations

3.5.1. KvipPostSend

Synopsis

```
KVIP_RETURN
        KvipPostSend(
                IN      KVIP_VI_HANDLE      ViHandle,
                IN      KVIP_DESCRIPTOR     *DescriptorPtr,
                IN      KVIP_MEM_HANDLE     MemoryHandle
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

DescriptorPtr: Pointer to a Descriptor to be posted on the send queue.

MemoryHandle: The handle for the memory region of the Descriptor being posted.

Description

KvipPostSend adds a Descriptor to the tail of the send queue of a VI, notifies the NIC that a new Descriptor is available, and returns immediately.

Returns

KVIP_SUCCESS – The send Descriptor was successfully posted.

3.5.2. KvipSendDone

Synopsis

```
KVIP_RETURN
        KvipSendDone(
                IN      KVIP_VI_HANDLE      ViHandle,
                OUT     KVIP_DESCRIPTOR     **DescriptorPtr
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

DescriptorPtr: Address of the Descriptor that has completed, if any.

Description

KvipSendDone checks the Descriptor at the head of the send queue to see if it has been marked complete. If the operation has completed, the Descriptor is removed from the head of the queue and the address of the Descriptor is returned. A KVIP_DESCRIPTOR_ERROR is returned if the operation completed with errors returned in the Descriptor status or the send queue is empty. If the send queue is empty, DescriptorPtr is set to NULL. *KvipSendDone* is a non-blocking call.

Returns

KVIP_SUCCESS – A completed Descriptor was returned with a successful completion status.

KVIP_DESCRIPTOR_ERROR - If the send queue is empty, the Descriptor pointer is set to NULL, otherwise, a completed Descriptor is returned with an error completion status.

KVIP_NOT_DONE – No completed Descriptor was found.

3.5.3. KvipRearmSend

Synopsis

```
KVIP_RETURN
        KvipRearmSend(
                IN      KVIP_VI_HANDLE      ViHandle
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

Description

KvipRearmSend resets the SendWaitObject referred to by the ViHandle. Note: interrupts are enabled, but it is up to the user to reset the event that is associated with the object.

KvipRearmSend returns a KVIP_ERROR_RESOURCE if the send Work Queue is associated with a Completion Queue. If the send Work Queue is not associated with a SendWaitObject, KVIP_INVALID_PARAMETER is returned.

Returns

KVIP_SUCCESS – The rearm function was successful.

KVIP_INVALID_PARAMETER – The Work Queue is not associated with a SendWaitObject.

KVIP_ERROR_RESOURCE - The Work Queue is associated with a Completion Queue or the operation failed due to insufficient resources.

3.5.4. KvipPostRecv

Synopsis

```
KVIP_RETURN
        KvipPostRecv(
                IN      KVIP_VI_HANDLE      ViHandle,
                IN      KVIP_DESCRIPTOR     *DescriptorPtr,
                IN      KVIP_MEM_HANDLE     MemoryHandle
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

DescriptorPtr: Pointer to a Descriptor to be posted on the receive queue.

MemoryHandle: The handle for the memory region of the Descriptor being posted.

Description

KvipPostRecv adds a Descriptor to the tail of the receive queue of the specified VI, notifies the NIC that a new Descriptor is available, and returns immediately.

Returns

KVIP_SUCCESS – The receive Descriptor was successfully posted.

3.5.5. KvipRecvDone

Synopsis

```
KVIP_RETURN
        KvipRecvDone(
                IN      KVIP_VI_HANDLE      ViHandle,
                OUT     KVIP_DESCRIPTOR     **DescriptorPtr
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

DescriptorPtr: Address of the Descriptor that has completed, if any.

Description

KvipRecvDone checks the Descriptor on the head of the receive queue to see if it has been marked complete. If the receive has completed, the Descriptor is removed from the head of the queue and the address of the Descriptor is returned. A KVIP_DESCRIPTOR_ERROR is returned if the operation completed with errors returned in the Descriptor status or the receive queue is empty. If the receive queue is empty, DescriptorPtr is set to NULL. *KvipRecvDone* is a non-blocking call.

Returns

KVIP_SUCCESS – A completed receive Descriptor was returned with a successful completion status.

KVIP_DESCRIPTOR_ERROR - If the receive queue is empty, the Descriptor pointer is set to NULL, otherwise a completed Descriptor is returned with an error completion status.

KVIP_NOT_DONE – No completed Descriptor was found.

3.5.6. KvipRearmRecv

Synopsis

```
KVIP_RETURN
    KvipRearmRecv(
        IN      KVIP_VI_HANDLE      ViHandle
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

Description

KvipRearmRecv resets the RecvWaitObject referred to by the ViHandle. Note: interrupts are enabled, but it is up to the user to reset the event that is associated with the object.

KvipRearmRecv returns a KVIP_ERROR_RESOURCE if the receive Work Queue is associated with a Completion Queue. If the receive Work Queue is not associated with a RecvWaitObject, KVIP_INVALID_PARAMETER is returned.

Returns

KVIP_SUCCESS – The rearm function was successful.

KVIP_INVALID_PARAMETER – The receive Work Queue is not associated with a RecvWaitObject.

KVIP_ERROR_RESOURCE - The Work Queue is associated with a Completion Queue or the operation failed due to insufficient resources.

3.5.7. KvipCQDone

Synopsis

```
KVIP_RETURN
        KvipCQDone(
                IN      KVIP_CQ_HANDLE      CQHandle,
                OUT     KVIP_VI_HANDLE      *ViHandle,
                OUT     KVIP_BOOLEAN        *RecvQueue
        )
```

Parameters

CQHandle: The handle of the Completion Queue.

ViHandle: The handle of the VI endpoint associated with the completion, if the return status indicates success. Undefined otherwise.

RecvQueue: If KVIP_TRUE, indicates that the completion was associated with the receive queue of the VI. If KVIP_FALSE, indicates that the completion was associated with the send queue of the VI. Undefined if the returned status does not indicate success.

Description

KvipCQDone polls the specified Completion Queue for a completion entry (a completed operation). If a completion entry is found, it returns the VI handle, along with a flag to indicate whether the completed Descriptor resides on the send or receive queue. *KvipCQDone* is a non-blocking call.

It is up to the calling process to subsequently invoke the appropriate function to actually de-queue the completed Descriptor. The completed Descriptor may only be de-queued by the function *KvipSendDone* or *KvipRecvDone*. *KvipCQDone* only dequeues the completion entry from the Completion Queue.

It is possible for a process to have multiple threads, some of which are waiting for completions on a Completion Queue, and others polling the Work Queues of an associated VI. In this case, the caller must be prepared for the case where the Completion Queue indicated a completion, but a subsequent call to de-queue the Descriptor fails.

Returns

KVIP_SUCCESS – A completion entry was found on the Completion Queue.

KVIP_NOT_DONE – No completion entries are on the Completion Queue.

3.5.8. KvipRearmCQ

Synopsis

```
KVIP_RETURN
        KvipRearmCQ(
                IN      KVIP_CQ_HANDLE      CQHandle
        )
```

Parameters

CQHandle: The Handle of the Completion Queue.

Description

KvipRearmCQ resets the CQWaitObject referred to by the CQHandle. Note: interrupts are enabled, but it is up to the user to reset the event that is associated with the object.

A KVIP_INVALID_PARAMETER error is returned if the Completion Queue is not associated with a CQWaitObject.

Returns

KVIP_SUCCESS – The rearm function was successful.

KVIP_INVALID_PARAMETER – The Completion Queue is not associated with a CQWaitObject.

KVIP_ERROR_RESOURCE - The operation could not be completed due to insufficient resources.

3.6. Completion Queue Management

3.6.1. KvipCreateCQ

Synopsis

```
KVIP_RETURN
        KvipCreateCQ(
                IN      KVIP_NIC_HANDLE     NicHandle,
                IN      KVIP_ULONG          EntryCount,
                IN      KVIP_WAIT_OBJECT    *CQWaitObject,
                OUT     KVIP_CQ_HANDLE      *CQHandle
        )
```

Parameters

NicHandle: The handle of the associated NIC.

EntryCount: The number of completion entries that this Completion Queue will hold.

CQWaitObject: An object that is signaled by KVIPL when a completed descriptor is present on this Completion Queue.

CQHandle: Returned to the caller. The handle of the newly created Completion Queue.

Description

KvipCreateCQ creates a new Completion Queue. The caller must specify the minimum number of completion entries that the queue must contain. If successful, it returns a handle to the newly created Completion Queue. A Completion queue is created with at least the specified number of completion entries. To avoid dropped completion notifications, applications should make sure that the number of operations posted on send/receive queues attached to a completion queue does not exceed the completion queue capacity at any time. A common technique to deal with this in multi-threaded environments is to use atomic increment/decrement variables to keep track of the available space in completion queues.

The CQWaitObject parameter provides a means for KVIPL client applications to be notified when completed descriptors are available on any send or receive VI Work Queues associated with this Completion Queue. The application has the option to wait, use only polling or poll to check status while waiting. The application can wait by waiting for the wait objects to be signaled using standard kernel synchronization routines. The application polls by checking the completion status of Work Queues associated with this Completion Queue using the *KvipCQDone* function.

The *KvipRearmCQ* function resets the CQWaitObject to allow the application to wait on the object again after it handles the previous signal. A recommended programming model is to call *KvipRearmCQ* to reset the wait object after processing all of the completed descriptors from the associated Completion Queue.

Returns

KVIP_SUCCESS – A new Completion Queue was successfully created.

KVIP_ERROR_RESOURCE – The Completion Queue could not be created due to insufficient resources.

3.6.2. KvipDestroyCQ

Synopsis

```
KVIP_RETURN
    KvipDestroyCQ(
        IN    KVIP_CQ_HANDLE    CQHandle
    )
```

Parameters

CQHandle: The handle of the Completion Queue to be destroyed.

Description

KvipDestroyCQ destroys a specified Completion Queue. If any VI Work Queues are associated with the Completion Queue, the Completion Queue is not destroyed and an error is returned.

Returns

KVIP_SUCCESS – The Completion Queue was successfully destroyed.

KVIP_ERROR_RESOURCE – The Completion Queue could not be destroyed because the Work Queues of one or more VI instances are still associated with it.

3.6.3. KvipResizeCQ

Synopsis

```
KVIP_RETURN
    KvipResizeCQ(
        IN    KVIP_CQ_HANDLE    CQHandle,
        IN    KVIP_ULONG        EntryCount
    )
```

Parameters

CQHandle: The handle of the Completion Queue to be resized.

EntryCount: The new number of completion entries that the Completion Queue must hold.

Description

KvipResizeCQ modifies the size of a specified Completion Queue by specifying the new minimum number of completion entries that it must hold. This function is useful when the potential number of completion entries that could be placed on this queue changes dynamically.

Returns

KVIP_SUCCESS – The Completion Queue was successfully resized.

KVIP_ERROR_RESOURCE – The Completion Queue could not be resized because of insufficient resources.

3.7. Querying Information

3.7.1. KvipQueryNic

Synopsis

```
KVIP_RETURN
    KvipQueryNic(
        IN    KVIP_NIC_HANDLE        NicHandle,
        OUT   KVIP_NIC_ATTRIBUTES    *NicAttribs
    )
```

Parameters

NicHandle: The handle of a VI NIC.

NicAttribs: Returned to the caller, contains NIC-specific information.

Description

KvipQueryNic returns information for a specific NIC instance. The information is returned in the NicAttribs data structure.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.2. KvipSetViAttributes

Synopsis

```
KVIP_RETURN
    KvipSetViAttributes(
        IN    KVIP_VI_HANDLE         ViHandle,
        IN    KVIP_VI_ATTRIBUTES     *ViAttribs
    )
```

Parameters

ViHandle: The handle of a VI instance.

ViAttribs: The attributes to be set for the VI.

Description

KvipSetViAttributes attempts to modify the attributes of a VI instance. If the VI Provider does not support the requested attributes, or if the VI is in a state that does not allow the attributes to be modified, then it returns an error.

Changing VI attributes is valid only when the VI is in the Idle state. An error is returned if *KvipSetViAttributes* is called while the VI is in any other state. When an error is returned, all of the attributes remain unchanged. For instance, if only one of many requested attributes is invalid or not supported, an error is returned and none of the new attributes requested are updated.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_INVALID_STATE – The VI is not in a state where the attributes can be modified.

KVIP_INVALID_ RELIABILITY_LEVEL – The requested reliability level attribute was invalid or not supported.

KVIP_INVALID_MTU – The maximum transfer size attribute was invalid or not supported.

KVIP_INVALID_QOS – The quality of service attribute was invalid or not supported.

KVIP_INVALID_PTAG – The protection tag attribute was invalid.

KVIP_INVALID_RDMAREAD – The attributes requested support for RDMA Read, but the VI Provider does not support it.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.3. KvipQueryVi

Synopsis

```
KVIP_RETURN
    KvipQueryVi(
        IN    KVIP_VI_HANDLE        ViHandle,
        OUT   KVIP_VI_STATE         *State,
        OUT   KVIP_VI_ATTRIBUTES    *ViAttribs,
        OUT   KVIP_BOOLEAN          *ViSendQEmpty,
        OUT   KVIP_BOOLEAN          *ViRecvQEmpty
    )
```

Parameters

ViHandle: The handle of a VI instance.

State: The current state of the VI.

ViAttribs: Returned to caller, contains VI-specific information.

ViSendQEmpty: If KVIP_TRUE, the send queue is empty.

ViRecvQEmpty: If KVIP_TRUE, the receive queue is empty.

Description

KvipQueryVi returns information for a specific VI instance. The VI Attributes data structure and the current VI State are returned.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.4. KvipSetMemAttributes

Synopsis

```
KVIP_RETURN
        KvipSetMemAttributes(
                IN      KVIP_NIC_HANDLE         NicHandle,
                IN      KVIP_PVOID              Address,
                IN      KVIP_MEM_HANDLE         MemHandle,
                IN      KVIP_MEM_ATTRIBUTES     *MemAttribs
        )
```

Parameters

NicHandle: The handle of the NIC where the memory region is registered.

Address: The base address of the memory region.

MemHandle: The handle of the memory region.

MemAttribs: The memory attributes to set for this memory region.

Description

KvipSetMemAttributes modifies the attributes of a registered memory region. If the VI Provider does not support the requested attribute, it returns an error. Modifying the attributes of a memory region, while a data transfer operation is in progress that refers to that memory region, can result in undefined behavior, and should be avoided by the VI Consumer.

When an error is returned, all of the attributes remain unchanged. For instance if only one of many requested attributes is invalid or not supported, an error is returned and none of the new attributes requested are updated.

To modify the attributes of a registered physical memory region, set the Address parameter to NULL and MemHandle to the memory handle returned by *KvipRegisterPhysPages*.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_INVALID_PTAG – The protection tag attribute was invalid.

KVIP_INVALID_RDMAREAD – The attributes requested support for RDMA Read, but the VI Provider does not support it.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.5. KvipQueryMem

Synopsis

```
KVIP_RETURN
        KvipQueryMem(
                IN      KVIP_NIC_HANDLE         NicHandle,
                IN      KVIP_PVOID              Address,
                IN      KVIP_MEM_HANDLE         MemHandle,
                OUT     KVIP_MEM_ATTRIBUTES     *MemAttribs
        )
```

Parameters

NicHandle: The handle of the NIC where the memory region is registered.

Address: The base address of the memory region.

MemHandle: The handle of a memory region.

MemAttribs: The memory attributes of this memory region.

Description

KvipQueryMem returns the attributes of a registered memory region to the caller.

To query the attributes of a registered physical memory region, set the Address parameter to NULL and MemHandle to the memory handle returned by *KvipRegisterPhysPages*.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.6. KvipQuerySystemManagementInfo

Synopsis

```
KVIP_RETURN
    KvipQuerySystemManagementInfo(
        IN      KVIP_NIC_HANDLE     NicHandle,
        IN      KVIP_ULONG          InfoType,
        IN OUT  KVIP_PVOID          SysManInfo
    )
```

Parameters

NicHandle: The handle of a VI NIC.

InfoType: Specifies a particular piece of system management information.

SysManInfo: Pointer to a system management information structure corresponding to the requested InfoType.

Description

KvipQuerySystemManagementInfo returns system management information about the specified NIC. The InfoType parameter allows the caller to specify specific pieces of information. The types of information that can be retrieved are VI Provider specific. The content of the System Management Information Structure is VI Provider specific.

Returns

KVIP_SUCCESS – Operation completed successfully.

3.8. Error handling

3.8.1. KvipErrorNotificationDone

Synopsis

```
KVIP_RETURN
    KvipErrorNotificationDone(
        IN    KVIP_NIC_HANDLE          NicHandle,
        OUT   KVIP_ERROR_DESCRIPTOR    **ErrorDescriptor
    )
```

Parameters

NicHandle: Handle of the NIC

ErrorDescriptor: The Error Descriptor

Description

KvipErrorNotificationDone is a non-blocking call that checks to see if the NIC has posted any ErrorDescriptors. If an ErrorDescriptor is found, the Descriptor is removed from the head of the queue and a pointer to the Descriptor is returned. If the queue is empty, the ErrorDescriptor parameter is set to NULL, the memory pointed to by ErrorDescriptor is not altered and a KVIP_NOT_DONE error is returned.

The client is responsible for allocating memory for the ErrorDescriptor before making this call, otherwise the error, KVIP_INVALID_PARAMETER, will be returned.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

KVIP_NOT_DONE - No ErrorDescriptor was found.

KVIP_INVALID_PARAMETER - Memory for the ErrorDescriptor was not allocated by the caller.

3.8.2. KvipRearmErrorNotification

Synopsis

```
KVIP_RETURN
    KvipRearmErrorNotification (
        IN    KVIP_NIC_HANDLE          NicHandle
    )
```

Parameters

NicHandle: Handle of the NIC

Description

KvipRearmErrorNotification resets the ErrorNotificationWaitObject associated with the NIC referred to by NicHandle.

Note: Interrupts are enabled but it is up to the user to reset the event that is associated with the object.

Returns

KVIP_SUCCESS – Operation completed successfully.

KVIP_INVALID_PARAM - The NicHandle was invalid.

KVIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.9. VI Context

3.9.1. KvipSetViContext

Synopsis

```
KVIP_RETURN
    KvipSetViContext(
        IN    KVIP_VI_HANDLE    ViHandle,
        IN    KVIP_PVOID        ViContext
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

ViContext: A client assigned and managed context that can be retrieved by calling *KvipGetViContext.*

Description

KvipSetViContext provides a mechanism for KVIPL client applications to assign a user-defined context to an existing VI. To retrieve the context, the call *KvipGetViContext* has been provided.

If ViHandle is NULL, a VIP_INVALID_PARAMETER error is returned.

Returns

KVIP_SUCCESS - The connection request was successfully posted by the server.

KVIP_INVALID_PARAMETER - One of the parameters was invalid.

3.9.2. KvipGetViContext

Synopsis

```
KVIP_RETURN
    KvipGetViContext(
        IN    KVIP_VI_HANDLE    ViHandle,
        OUT   KVIP_PVOID        *ViContext
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

ViContext: Pointer to the context previously assigned to this VI by *KvipSetViContext.*

Description

KvipGetViContext provides a mechanism for KVIPL client applications to extract a context that was assigned to a VI by calling KvipSetViContext.

If ViHandle is NULL, a VIP_INVALID_PARAMETER error is returned.

If a context has not been previously assigned to this VI by calling *KvipSetViContext*, VIP_ERROR_RESOURCE is returned.

Returns

KVIP_SUCCESS - The connection request was successfully posted by the server.

KVIP_INVALID_PARAMETER - One of the parameters was invalid.

KVIP_ERROR_RESOURCE - A context was not previously assigned to this VI.

4. Data Structures and Values

This section contains data structures and values that are specific to KVIPL. Data structures that are shared with VIPL are defined in the Intel 1.0 VIPL Developer's Guide.

5. Application Notes

This section contains recommendations, example usage and warnings for application developers using the KVIPL interface.

5.1. Polling model

Figure 1 depicts an example of how an application can utilize the polling model for a single request. It is possible to have multiple requests outstanding, but to simplify the figure only one request is shown. The diagram has been split to show the control flow between the application and the KVIPL library.

This model works best when an application can continue to make progress after the request is posted without having to wait for the request to complete.

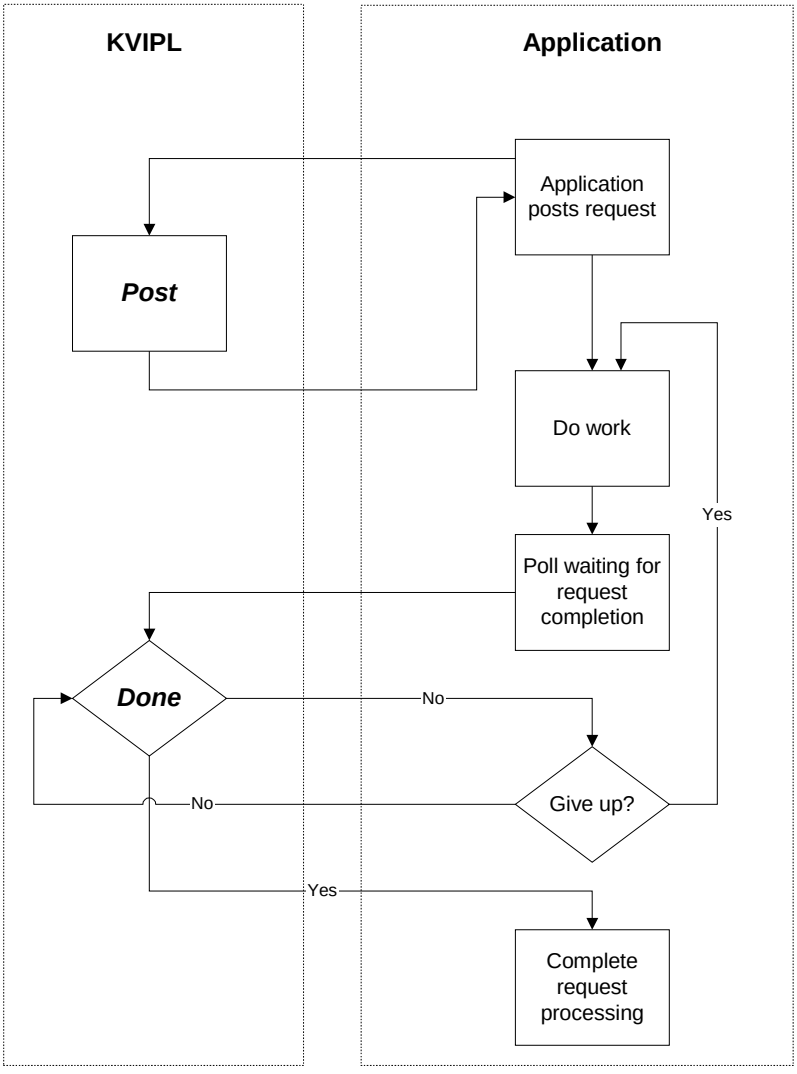

Figure 1: Polling Model

5.2. Wait model

Figure 2 is an example of how an application could use a wait model for a single request. Note that it is possible to have multiple requests outstanding at once, but to simplify the diagram only one request is shown. This diagram has been split to show the control flow between the application and the KVIPL library.

The example shows an application that posts a request and does work until it must wait for the request to complete to do additional work. The polling is done prior to and just after the *Kvip*Rearm* routine is called in case the request completed in the time window where the event signal would not be generated. Note that when the event signal occurs, the *Kvip*Done* routine must be called to be sure that the request has completed and to retrieve the attributes before the request processing can be completed.

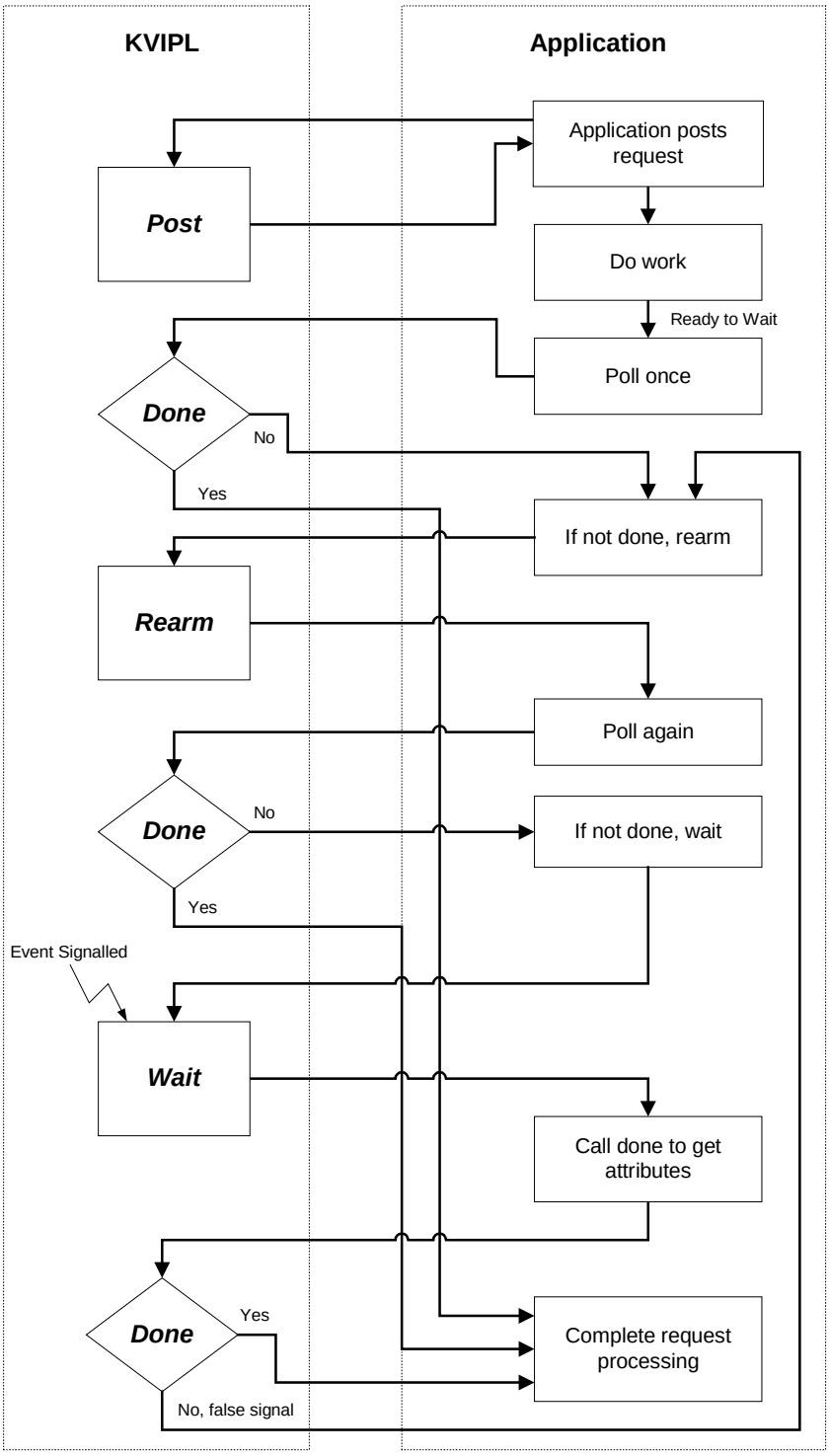

Figure 2: Wait model

6. VI Provider Notes

6.1. NT specifics

6.2. UNIX specifics

7. Appendix A - Include Files

7.1. kvipl.h

```
/*++

Copyright (c) 1998 Intel Corp. All Rights Reserved.

VIRTUAL INTERFACE ARCHITECTURE KVIPL.H

Intel Corporation hereby grants a non-exclusive license under Intel's
copyright to copy, modify and distribute this software for any purpose
and without fee, provided that the above copyright notice and the
following paragraphs appear on all copies.

Intel Corporation makes no representation that this source code
is correct or is an accurate representation of any standard.

IN NO EVENT SHALL INTEL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, OR
SPECULATIVE DAMAGES, (INCLUDING WITHOUT LIMITING THE FORGOING,
CONSEQUENTIAL, INCIDENTAL AND SPECIAL DAMAGES) INCLUDING, BUT NOT
LIMITED TO INFRINGEMENT, LOSS OF USE, BUSINESS INTERRUPTIONS, AND LOSS
OF PROFITS, IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE
POSSIBILITY OF ANY SUCH DAMAGES.

INTEL CORPORATION SPECIFICALLY DISCLAIMS ANY WARRANTIES INCLUDING, BUT
NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR
A PARTICULAR PURPOSE AND NON-INFRINGEMENT.  THE SOFTWARE PROVIDED
HEREUNDER IS ON AN "AS IS" BASIS AND INTEL CORPORATION HAS NO
OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS OR
MODIFICATIONS.

Module Name:
      kvipl.h

Abstract:

  kvipl.h contains the complete kernel user interface to VI.

--*/


#ifndef _KVIPL_10_H
#define _KVIPL_10_H

/*
 * Check for operating environment
 */

#ifdef WIN32 // TBD
#define EXPORT    DLLEXPORT
#endif
```

```
/*
 * Include vipl.h to reuse the original declarations of most
 * data structures and functions
 */

#include <vipl.h>

#if __cplusplus
extern "C" {
#endif

/*************************************************************************
 * Data types
*************************************************************************/

/*
 * Generic types for portability
 */

typedef VIP_UCHAR       KVIP_UCHAR;
typedef VIP_CHAR        KVIP_CHAR;
typedef WCHAR           KVIP_WCHAR;

typedef VIP_USHORT      KVIP_USHORT;
typedef VIP_ULONG       KVIP_ULONG;

typedef VIP_BOOLEAN     KVIP_BOOLEAN;
typedef VIP_PVOID       KVIP_PVOID;

/*
 * The following are NT specific
 */

typedef VIP_UINT64      KVIP_UINT64;
typedef VIP_UINT32      KVIP_UINT32;
typedef VIP_UINT16      KVIP_UINT16;
typedef VIP_UINT8       KVIP_UINT8;


/*
 * Constants used for KVIP_BOOLEAN
 */

#define KVIP_TRUE       VIP_TRUE
#define KVIP_FALSE      VIP_FALSE

/*
 * OS specific constructs
 */

typedef KEVENT KVIP_WAIT_OBJECT;
```

```
/*
 * Opaque handle types
 */

typedef VIP_NIC_HANDLE              KVIP_NIC_HANDLE;
typedef VIP_VI_HANDLE               KVIP_VI_HANDLE;
typedef VIP_CQ_HANDLE               KVIP_CQ_HANDLE;
typedef VIP_MEM_HANDLE              KVIP_MEM_HANDLE;
typedef VIP_PROTECTION_HANDLE       KVIP_PROTECTION_HANDLE;
typedef VIP_QOS                     KVIP_QOS;

/*
 * Kvip specific opaque handle
 */

typedef VIP_PVOID KVIP_CONN_HANDLE;
typedef VIP_PVOID KVIP_PHYS_ADDRESS;

/*
 * Timeout values
 */

#define KVIP_INFINITE   VIP_INFINITE

/*
 * VI reliability levels
 */

typedef VIP_RELIABILITY_LEVEL            KVIP_RELIABILITY_LEVEL;
#define KVIP_SERVICE_UNRELIABLE          VIP_SERVICE_UNRELIABLE
#define KVIP_SERVICE_RELIABLE_DELIVERY   VIP_SERVICE_RELIABLE_DELIVERY
#define KVIP_SERVICE_RELIABLE_RECEPTION  VIP_SERVICE_RELIABLE_RECEPTION

/*
 * Net address
 */

typedef VIP_NET_ADDRESS KVIP_NET_ADDRESS;

/*
 * NIC attributes structure
 */

typedef VIP_NIC_ATTRIBUTES    KVIP_NIC_ATTRIBUTES;

/*
 * Memory attributes strcuture
 */

typedef enum {
    KVIP_RESOURCE_NIC,
    KVIP_RESOURCE_VI,
    KVIP_RESOURCE_CQ,
    KVIP_RESOURCE_DESCRIPTOR,
} KVIP_RESOURCE_CODE;
```

```
typedef enum {
    KVIP_ERROR_POST_DESC,
    KVIP_ERROR_CONN_LOST,
    KVIP_ERROR_RECVQ_EMPTY,
    KVIP_ERROR_VI_OVERRUN,
    KVIP_ERROR_RDMAW_PROT,
    KVIP_ERROR_RDMAW_DATA,
    KVIP_ERROR_RDMAW_ABORT,
    KVIP_ERROR_RDMAR_PROT,
    KVIP_ERROR_COMP_PROT,
    KVIP_ERROR_RDMA_TRANSPORT,
    KVIP_ERROR_CATASTROPHIC,
} KVIP_ERROR_CODE;

typedef VIP_ERROR_DESCRIPTOR  KVIP_ERROR_DESCRIPTOR;

typedef VIP_MEM_ATTRIBUTES    KVIP_MEM_ATTRIBUTES;

/*
 * VI states
 */

typedef enum {
    KVIP_STATE_IDLE,
    KVIP_STATE_CONNECTED,
    KVIP_STATE_CONNECT_PENDING,
    KVIP_STATE_ERROR,
} KVIP_VI_STATE;

/*
 * VI attributes structure
 */

typedef VIP_VI_ATTRIBUTES     KVIP_VI_ATTRIBUTES;

/*
 * KVIP function return codes
 */

typedef enum {
    KVIP_SUCCESS,
    KVIP_NOT_DONE,
    KVIP_INVALID_PARAMETER,
    KVIP_ERROR_RESOURCE,
    KVIP_TIMEOUT,
    KVIP_REJECT,
    KVIP_INVALID_RELIABILITY_LEVEL,
    KVIP_INVALID_MTU,
    KVIP_INVALID_QOS,
    KVIP_INVALID_PTAG,
    KVIP_INVALID_RDMAREAD,
    KVIP_DESCRIPTOR_ERROR,
    KVIP_INVALID_STATE,
    KVIP_ERROR_NAMESERVICE,
    KVIP_NO_MATCH,
    KVIP_NOT_REACHABLE,
} KVIP_RETURN;
```

```
/*
 * Bit field macros for the descriptor control segment: Control field
 */

#define KVIP_CONTROL_OP_SENDRECV          VIP_CONTROL_OP_SENDRECV
#define KVIP_CONTROL_OP_RDMAWRITE         VIP_CONTROL_OP_RDMAWRITE
#define KVIP_CONTROL_OP_RDMAREAD          VIP_CONTROL_OP_RDMAREAD
#define KVIP_CONTROL_OP_RESERVED          VIP_CONTROL_OP_RESERVED
#define KVIP_CONTROL_OP_MASK              VIP_CONTROL_OP_MASK
#define KVIP_CONTROL_IMMEDIATE            VIP_CONTROL_IMMEDIATE
#define KVIP_CONTROL_QFENCE               VIP_CONTROL_QFENCE
#define KVIP_CONTROL_RESERVED             VIP_CONTROL_RESERVED

/*
 * Bit field macros for the descriptor control segment: Status field
 */

#define KVIP_STATUS_DONE                  VIP_STATUS_DONE
#define KVIP_STATUS_DONE                  VIP_STATUS_DONE
#define KVIP_STATUS_FORMAT_ERROR          VIP_STATUS_FORMAT_ERROR
#define KVIP_STATUS_PROTECTION_ERROR      VIP_STATUS_PROTECTION_ERROR
#define KVIP_STATUS_LENGTH_ERROR          VIP_STATUS_LENGTH_ERROR
#define KVIP_STATUS_PARTIAL_ERROR         VIP_STATUS_PARTIAL_ERROR
#define KVIP_STATUS_DESC_FLUSHED_ERROR    VIP_STATUS_DESC_FLUSHED_ERROR
#define KVIP_STATUS_TRANSPORT_ERROR       VIP_STATUS_TRANSPORT_ERROR
#define KVIP_STATUS_RDMA_PROT_ERROR       VIP_STATUS_RDMA_PROT_ERROR
#define KVIP_STATUS_REMOTE_DESC_ERROR     VIP_STATUS_REMOTE_DESC_ERROR
#define KVIP_STATUS_ERROR_MASK            VIP_STATUS_ERROR_MASK

#define KVIP_STATUS_OP_SEND               VIP_STATUS_OP_SEND
#define KVIP_STATUS_OP_RECEIVE            VIP_STATUS_OP_RECEIVE
#define KVIP_STATUS_OP_RDMA_WRITE         VIP_STATUS_OP_RDMA_WRITE
#define KVIP_STATUS_OP_REMOTE_RDMA_WRITE  VIP_STATUS_OP_REMOTE_RDMA_WRITE
#define KVIP_STATUS_OP_RDMA_READ          VIP_STATUS_OP_RDMA_READ
#define KVIP_STATUS_OP_MASK               VIP_STATUS_OP_MASK
#define KVIP_STATUS_IMMEDIATE             VIP_STATUS_IMMEDIATE
#define KVIP_STATUS_RESERVED              VIP_STATUS_RESERVED

/*
 * Descriptor
 */

typedef     VIP_PVOID64                   KVIP_PVOID64;
typedef     VIP_CONTROL_SEGMENT           KVIP_CONTROL_SEGMENT;
typedef     VIP_ADDRESS_SEGMENT           KVIP_ADDRESS_SEGMENT;
typedef     VIP_DATA_SEGMENT              KVIP_DATA_SEGMENT;
typedef     VIP_DESCRIPTOR_SEGMENT        KVIP_DESCRIPTOR_SEGMENT;
typedef     VIP_DESCRIPTOR                KVIP_DESCRIPTOR;

#define     KVIP_DESCRIPTOR_ALIGNMENT     VIP_DESCRIPTOR_ALIGNMENT
```

```
/*********************************************************************
 * Functions
*********************************************************************/

/*
 * Hardware Connection
 */

EXPORT
KVIP_RETURN
KvipOpenNic( IN   KVIP_WCHAR const        *DeviceName,
             IN   KVIP_WAIT_OBJECT        *ErrorNotificationWaitObject,
             OUT  KVIP_NIC_HANDLE         *NicHandle );

EXPORT
KVIP_RETURN
KvipCloseNic( IN KVIP_NIC_HANDLE NicHandle );


/*
 * Endpoint Creation and Destruction
 */

EXPORT
KVIP_RETURN
KvipCreateVi ( IN  KVIP_NIC_HANDLE        NicHandle,
               IN  KVIP_VI_ATTRIBUTES     *ViAttribs,
               IN  KVIP_CQ_HANDLE         SendCQHandle,
               IN  KVIP_CQ_HANDLE         RecvCQHandle,
               IN  KVIP_WAIT_OBJECT       *SendWaitObject,
               IN  KVIP_WAIT_OBJECT       *RecvWaitObject,
               OUT KVIP_VI_HANDLE         *ViHandle );

EXPORT
KVIP_RETURN
KvipDestroyVi( IN KVIP_VI_HANDLE ViHandle );

/*
 * Connection Management
 */

EXPORT
KVIP_RETURN
KvipConnectServerRequest( IN  KVIP_NIC_HANDLE   NicHandle,
                          IN  KVIP_NET_ADDRESS  *LocalAddr,
                          IN  KVIP_ULONG        Timeout,
                          IN  KVIP_WAIT_OBJECT  *WaitObject,
                          OUT KVIP_CONN_HANDLE  *ConnHandle );

EXPORT
KVIP_RETURN
KvipConnectServerDone( IN  KVIP_CONN_HANDLE     ConnHandle,
                       OUT KVIP_NET_ADDRESS     *RemoteAddr,
                       OUT KVIP_VI_ATTRIBUTES   *RemoteViAtrribs );
```

```
EXPORT
KVIP_RETURN
KvipConnectClientRequest( IN  KVIP_VI_HANDLE    ViHandle,
                          IN  KVIP_NET_ADDRESS  *LocalAddr,
                          IN  KVIP_NET_ADDRESS  *RemoteAddr,
                          IN  KVIP_ULONG        Timeout,
                          IN  KVIP_WAIT_OBJECT  *WaitObject );

EXPORT
KVIP_RETURN
KvipConnectClientDone( IN  KVIP_VI_HANDLE       ViHandle,
                       OUT KVIP_VI_ATTRIBUTES   *RemoteViAttribs );

EXPORT
KVIP_RETURN
KvipConnectAccept( IN  KVIP_CONN_HANDLE         ConnHandle,
                   IN  KVIP_VI_HANDLE           ViHandle );

EXPORT
KVIP_RETURN
KvipConnectReject( IN KVIP_CONN_HANDLE ConnHandle );

EXPORT
KVIP_RETURN
KvipDisconnect( IN KVIP_VI_HANDLE ViHandle );

EXPORT
KVIP_RETURN
KvipConnectPeerRequest( IN  KVIP_VI_HANDLE      ViHandle,
                        IN  KVIP_NET_ADDRESS    *LocalAddr,
                        IN  KVIP_NET_ADDRESS    *RemoteAddr,
                        IN  KVIP_ULONG          Timeout,
                        IN  KVIP_WAIT_OBJECT    *WaitObject );

EXPORT
KVIP_RETURN
KvipConnectPeerDone( IN  KVIP_VI_HANDLE         ViHandle,
                     OUT KVIP_VI_ATTRIBUTES     *RemoteViAttribs );

/*
 * Memory Protection and Registration
 */

EXPORT
KVIP_RETURN
KvipCreatePtag( IN  KVIP_NIC_HANDLE             NicHandle,
                OUT  KVIP_PROTECTION_HANDLE     *Ptag );

EXPORT
KVIP_RETURN
KvipDestroyPtag( IN  KVIP_NIC_HANDLE            NicHandle,
                 IN  KVIP_PROTECTION_HANDLE     Ptag );
```

```
EXPORT
KVIP_RETURN
KvipRegisterMem( IN  KVIP_NIC_HANDLE            NicHandle,
                 IN  KVIP_PVOID                 VirtualAddress,
                 IN  KVIP_ULONG                 Length,
                 IN  KVIP_MEM_ATTRIBUTES        *MemAttribs,
                 OUT KVIP_MEM_HANDLE            *MemoryHandle );

EXPORT
VIP_RETURN
KvipDeregisterMem( IN  KVIP_NIC_HANDLE          NicHandle,
                   IN  KVIP_PVOID               VirtualAddress,
                   IN  KVIP_MEM_HANDLE          MemoryHandle );

EXPORT
KVIP_RETURN
KvipRegisterPhysPages( IN  KVIP_NIC_HANDLE      NicHandle,
                       IN  KVIP_PHYS_ADDRESS    PhysAddress[],
                       IN  KVIP_ULONG           NumPhysPages,
                       IN  KVIP_MEM_ATTRIBUTES  *MemAttribs,
                       OUT KVIP_MEM_HANDLE      *MemoryHandle );

EXPORT
KVIP_RETURN
KvipDeregisterPhysPages( IN  KVIP_NIC_HANDLE    NicHandle,
                         IN  KVIP_MEM_HANDLE    MemoryHandle );

/*
 * Data transfer and completion operations
 */

EXPORT
KVIP_RETURN
KvipPostSend( IN  KVIP_VI_HANDLE          ViHandle,
              IN  KVIP_DESCRIPTOR         *DescriptorPtr,
              IN  KVIP_MEM_HANDLE         MemoryHandle );

EXPORT
KVIP_RETURN
KvipSendDone( IN  KVIP_VI_HANDLE          ViHandle,
              OUT KVIP_DESCRIPTOR         **DescriptorPtr);

EXPORT
KVIP_RETURN
KvipRearmSend( IN KVIP_VI_HANDLE    ViHandle );

EXPORT
KVIP_RETURN
KvipPostRecv( IN  KVIP_VI_HANDLE    ViHandle,
              IN  KVIP_DESCRIPTOR   *DescriptorPtr,
              IN  KVIP_MEM_HANDLE   MemoryHandle );

EXPORT
KVIP_RETURN
KvipRecvDone( IN  KVIP_VI_HANDLE    ViHandle,
              OUT KVIP_DESCRIPTOR   **DescriptorPtr );
```

```
EXPORT
KVIP_RETURN
KvipRearmRecv( IN KVIP_VI_HANDLE    ViHandle );

EXPORT
KVIP_RETURN
KvipCQDone( IN  KVIP_CQ_HANDLE      CQHandle,
           OUT KVIP_VI_HANDLE      *ViHandle,
           OUT KVIP_BOOLEAN        *RecvQueue );

EXPORT
KVIP_RETURN
KvipRearmCQ( IN KVIP_CQ_HANDLE      CQHandle );

/*
 * Completion Queue Management
 */

EXPORT
KVIP_RETURN
KvipCreateCQ( IN  KVIP_NIC_HANDLE   NicHandle,
              IN  KVIP_ULONG        EntryCount,
              IN  KVIP_WAIT_OBJECT  *CQWaitObject,
              OUT KVIP_CQ_HANDLE    *CQHandle );

EXPORT
KVIP_RETURN
KvipDestroyCQ( IN KVIP_CQ_HANDLE    CQHandle );

EXPORT
KVIP_RETURN
KvipResizeCQ( IN  KVIP_CQ_HANDLE    CQHandle,
              IN  KVIP_ULONG        EntryCount );

/*
 * Querying information
 */

EXPORT
KVIP_RETURN
KvipQueryNic( IN  KVIP_NIC_HANDLE               NicHandle,
              OUT KVIP_NIC_ATTRIBUTES           *NicAttribs );

EXPORT
KVIP_RETURN
KvipSetViAttributes( IN KVIP_VI_HANDLE          ViHandle,
                     IN KVIP_VI_ATTRIBUTES      *ViAttribs );

EXPORT
KVIP_RETURN
KvipQueryVi( IN   KVIP_VI_HANDLE                ViHandle,
             OUT  KVIP_VI_STATE                 *State,
             OUT  KVIP_VI_ATTRIBUTES            *ViAttribs,
             OUT  KVIP_BOOLEAN                  *ViSendQEmpty,
             OUT  KVIP_BOOLEAN                  *ViRecvQEmpty );
```

```
EXPORT
KVIP_RETURN
KvipSetMemAttributes( IN  KVIP_NIC_HANDLE       NicHandle,
                      IN  KVIP_PVOID            Address,
                      IN  KVIP_MEM_HANDLE       MemHandle,
                      IN  KVIP_MEM_ATTRIBUTES   *MemAttribs );

EXPORT
KVIP_RETURN
KvipQueryMem( IN  KVIP_NIC_HANDLE               NicHandle,
              IN  KVIP_PVOID                    Address,
              IN  KVIP_MEM_HANDLE               MemHandle,
              OUT KVIP_MEM_ATTRIBUTES           *MemAttribs );

EXPORT
KVIP_RETURN
KvipQuerySystemManagementInfo( IN     KVIP_NIC_HANDLE NicHandle,
                               IN     KVIP_ULONG      InfoType,
                               IN OUT KVIP_PVOID      SysManInfo );

/*
 * Error handling
 */

EXPORT
KVIP_RETURN
KvipErrorNotificationDone(IN   KVIP_NIC_HANDLE        NicHandle,
                          OUT KVIP_ERROR_DESCRIPTOR **ErrorDescriptor
);

EXPORT
KVIP_RETURN
KvipRearmErrorNotification( IN KVIP_NIC_HANDLE  NicHandle );

#if __cplusplus
};
#endif

#endif
```

7.2. kvipl.def

```
LIBRARY  kvipl

EXPORTS
  KvipOpenNic                     @1
  KvipCloseNic                    @2
  KvipCreateVi                    @3
  KvipDestroyVi                   @4
  KvipConnectServerRequest        @5
  KvipConnectServerDone           @6
  KvipConnectClientRequest        @7
  KvipConnectClientDone           @8
  KvipConnectAccept               @9
  KvipConnectReject               @10
  KvipDisconnect                  @11
  KvipConnectPeerRequest          @12
  KvipConnectPeerDone             @13
  KvipCreatePtag                  @14
  KvipDestroyPtag                 @15
  KvipRegisterMem                 @16
  KvipDeregisterMem               @17
  KvipRegisterPhysPages           @18
  KvipDeregisterPhysPages         @19
  KvipPostSend                    @20
  KvipSendDone                    @21
  KvipRearmSend                   @22
  KvipPostRecv                    @23
  KvipRecvDone                    @24
  KvipRearmRecv                   @25
  KvipCQDone                      @26
  KvipRearmCQ                     @27
  KvipCreateCQ                    @28
  KvipDestroyCQ                   @29
  KvipResizeCQ                    @30
  KvipQueryNic                    @31
  KvipSetViAttributes             @32
  KvipQueryVi                     @33
  KvipSetMemAttributes            @34
  KvipQueryMem                    @35
  KvipQuerySystemManagementInfo   @36
  KvipErrorNotificationDone       @37
  KvipRearmErrorNotification      @38
```